

Type homogeneity is not a restriction for safe recursion schemes

William Blum

January 10, 2017

Abstract

Knapik *et al.* introduced the *safety restriction* which constrains both the types and syntax of the production rules defining a higher-order recursion scheme [10]. This restriction gives rise to an equi-expressivity result between order- n pushdown automata and order- n safe recursion schemes, when such devices are used as tree generators.

We show that the typing constraint of safety, called *homogeneity*, is unnecessary in the sense that imposing the syntactic restriction alone is sufficient to prove the equi-expressivity result for trees.

1 Introduction

The concept of *safety* was introduced by Knapik *et al.* in the context of higher-order recursion schemes and studied further in [10, 7]. In the original paper they show that, when used as tree generators, safe recursion schemes are equivalent to safe pushdown automata [10]. In its original definition, safety consists of two fairly technical constraints:

1. **Type-Homogeneity** which imposes a type constraint on the rules of the recursion scheme;
2. **A syntactic restriction** on the relative order of sub-terms occurring in the right hand side of the recursion scheme rules.

Knapik *et al.*'s result begged a question: what is the automaton equivalent of ‘unsafe’ recursion schemes? This question was answered a few years later in a paper introducing a new kind of automata called Higher-Order Collapsible Pushdown Automata (CPDA), which is shown to be equivalent to (possibly unsafe) higher-order recursion schemes [9]. More specifically they describe an algorithm that, given an order- n recursive schemes G , constructs an equivalent order- n CPDA (in the sense that they both generate the same tree), and conversely. We will refer to them as the *HMOS transformation* procedures.

The safety restriction was further studied in the context of the lambda calculus in the author’s D.Phil. thesis [2]. Somewhat surprisingly, in the lambda calculus setting, the type-homogeneity constraint is not necessary to define a useful calculus. One can define a notion of safe simply-typed lambda calculus by imposing solely the syntactic constraint above, that we name ‘*incremental-binding*’, onto the standard simply-typed lambda calculus. The author’s D.Phil.

thesis gives expressivity results and presents a fully-abstract game model for this calculus [2].

This paper reconciles the above observation made in the lambda calculus with higher-order recursion schemes: we show that if a recursion scheme meets the syntactic criteria of the safety restriction, then regardless of whether the type homogeneity is met, the automaton constructed using the HMOS procedure from [9] can be converted into an equivalent order- n (non-collapsible) push-down automaton (PDA). Conversely, given an order- n PDA, the recursion scheme generated by the HMOS transformation induces an equivalent incrementally-bound and homogeneously-typed recursion scheme. Consequently, for generating trees, pushdown automata are just as expressive as incrementally-bound recursion schemes. (In what follows, by abuse of language, we will use “*incremental-binding*” to refer to the syntactic constraint 2. above, even though the concept of binders is foreign to recursion schemes.)

Further, composing the above two transformations yields a methods to convert any incrementally-bound recursion scheme into an equivalent *safe* one in the original sense (*i.e.*, incrementally-bound and type-homogeneous). Type-homogeneity is therefore not a proper restriction for safe recursion schemes. This generalizes Knapik *et al.*’s result on equi-expressivity of pushdown automata and *safe* recursion schemes [10].

Remark 1.1. The result presented in this paper was privately circulated for the first time in 2009 and shared on my personal website but was never published in a journal or conference [3]. The result was first conjectured in the author’s thesis in [2]. A partial proof was then privately circulated by the author in a 2008 note. Based on this note, Broadbent [5] adjusted the definition of *stack safety* to make the inductive proof work, which filled the remaining gap in the proof. The author then circulated an updated proof based on yet another definition of *stack safety* which is also the one used in the present paper.

2 Background

We first recall some basic notations and recall the definition of higher-order automata and higher-order recursion schemes from the literature [10, 8].

We consider simple types over a single atom o . We call Σ a **ranked alphabet** if each symbol $f \in \Sigma$ is assigned a type of the form $o \rightarrow \dots \rightarrow o \rightarrow o$. We write $\text{arity}(f)$ for any typed-term f to denote the arity of its type.

Given a set of typed symbols S , the set of **applicative terms** generated from S , is defined as the closure of S under the application rule: if m and n are two applicative terms of respective type $A \rightarrow B$ and A then (mn) is also an applicative term of type B .

A **Σ -labelled tree** is a possibly infinite tree where each nodes of the tree is labelled with a symbol in Σ with arity matching the number of children nodes in the tree.

2.1 Higher-order stacks and recursion schemes

Higher-order stacks The notion of higher-order stacks is defined inductively. We first fix the base alphabet as a finite set Γ . An *order-1* stack over Γ is a

sequence of elements of $\gamma \in \Gamma^*$. For $n \in \mathbb{N}$ an *order*-($n + 1$)-stack is defined as a stack of order- n stacks. The order of a stack s is written $\text{ord}(s)$. The i^{th} elements in a stack s , for $i \geq 0$, is denoted s_i so that a stack s consists of ordered elements s_1, \dots, s_k where $k \geq 0$ is called the length of the stack. The stack can then be represented by the notation $[s_1, \dots, s_k]$. The first element s_1 and last element s_k are respectively called the ‘bottom’ and ‘top’ elements of the stack. The empty stack of order 1 is denoted \perp_1 and represent the stack consisting of no element. We then define the empty $(n + 1)$ -stack \perp_{n+1} as the singleton sequence $[\perp_n]$.

The following operations are defined for an order-1 stack s :

$$\begin{aligned} \text{push}_1^a[s_1 \cdots s_m] &:= [s_1 \cdots s_m a] \\ \text{pop}_1[s_1 \cdots s_m s_{m+1}] &:= [s_1 \cdots s_m] \end{aligned}$$

and for an order- $(n + 1)$ stacks t :

$$\begin{aligned} \text{push}_{n+1}[t_1 \cdots t_m] &:= [t_1 \cdots t_m t_m] \\ \text{pop}_{n+1}[t_1 \cdots t_m t_{m+1}] &:= [t_1 \cdots t_m]. \end{aligned}$$

The top_i of a stack for $i \geq 1$ is defined as the top $(i - 1)$ -stack of s (so that pop_i returns s with its top $(i - 1)$ -stack removed). Formally, for $s = [s_1 \cdots s_{l+1}]$, $l \geq 0$ and $1 \leq i \leq \text{ord}(s)$:

$$\text{top}_i[s_1 \cdots s_{l+1}] := \begin{cases} s_{l+1} & \text{if } i = \text{ord}(s) \\ \text{top}_i s_{l+1} & \text{if } i < \text{ord}(s). \end{cases}$$

Following [9], we define the natural notion of stack prefix: for a given n -stack s , for any lower-level stack m occurring in s , the prefix $s_{\leq m}$ of s at m consists of the stack obtained from s by removing all the elements ‘above’ m using a succession of ‘pop’ operations.

Formally:

Definition 2.1 (Stack prefix). Let $s = [s_1 \cdots s_m]$ be a higher-order stack. Then $s_{\leq s_i}$ is defined as $[s_1 \cdots s_i]$ for $1 \leq i \leq m$; and for a stack t occurring in s_i we define $s_{\leq t}$ as $[s_1 \cdots s_{i \leq t}]$.

Similarly we define the strict stack prefix $s_{< s_i}$ and $s_{< t}$ respectively as $[s_1 \cdots s_{i-1}]$ and $[s_1 \cdots t_{< s_i}]$.

For any operation ϕ operating on a higher-order stack and $k \geq 1$ we denote ϕ^k the repeated application of ϕ exactly k times: $\phi^k s = \phi(\cdots(\phi s)\cdots)$ with k application of ϕ for any stack s .

Higher-order stacks with links We recall the notion of higher-order stack with links where each order-1 element is equipped with a link to another stack [8]:

Definition 2.2 (Higher order stack with links (or CPDA stack)). For a stack alphabet Γ and a distinguished bottom-of-stack symbol $\perp \in \Gamma$. An order-0 CPDA stack is just a stack symbol. An order- $(n + 1)$ CPDA stack s is a non-empty sequence (written $[s_1 \cdots s_l]$) of order n CPDA stacks such that every non- \perp symbol a that occurs in s has a link to a stack of some order k (where $0 \leq k \leq n$) situated below it in s ; we call the link a $(k + 1)$ -link.

An element of a higher order CPDA stack is written $a^{(j,k)}$ where $a \in \Gamma$ and the exponent (j, k) encodes the pointer associated to the stack symbol. Think of it as a shorthand for the iterated stack operation pop_j^k (i.e., k applications of pop_j). The value j is called the **order of the link**, and k is called the **height of the link** for $1 \leq j \leq n$ and $k \geq 1$, such that if $j = 1$ then $k = 1$.

Definition 2.3. Let s be a higher-order stack. We define $s^{(j)}$ as the operation that replaces every link occurring in s of the form (j, k) with $(j, k+1)$. Formally,

$$\begin{aligned} a^{(j,k)\langle j \rangle} &= a^{(j,k+1)} \\ a^{(j,k)\langle j' \rangle} &= a^{(j,k)} \quad \text{when } j \neq j', \\ [s_1 \dots s_p]^{(j)} &= [s_1^{(j)} \dots s_p^{(j)}]. \end{aligned}$$

This operation clearly commutes with prefixing. We will therefore write $s_{\leq m}^{(j)}$ as an abbreviation for $(s^{(j)})_{\leq m} = (s_{\leq m})^{(j)}$, for all stack s and stack-symbol m occurring in s .

The *top* and *pop* operations are defined identically to standard higher-order stacks. The *push* operation is defined similarly to higher-order stacks except that the push operation now assigns a pointer to every element pushed onto the stack. Also, in addition to the standard push and pop operations, the CPDA introduces a new *collapse* operation that ‘collapses’ a given stack to the target of the pointer associated with the top order-1 element in the stack. The idea is that if the top-1 element is a symbol a with a link to some $(j-1)$ -stack, then the *collapse* operation produces the same effect as successively performing k times the pop_j operation.

We reproduce here the definition of the *push* and *collapse* operations from [9]:

Definition 2.4 (Higher-order CPDA operations). We define CPDA operations in terms of the standard stack operations of an order- n PDA with an adequately defined alphabet encoding elements of the form $b^{(o,h)}$ where $b \in \Gamma$ and link indices $o, h \geq 0$. For $1 \leq i \leq \text{ord}(s)$ and $2 \leq j \leq \text{ord}(s)$:

$$\begin{aligned} push_1^{b,i} s &= push_1^{b^{(i,1)}} s \\ collapse s &= pop_o^h s \text{ where } top_1 s = a^{(o,h)} \\ push_j \underbrace{[s_1 \dots s_{l+1}]}_s &= \begin{cases} [s_1 \dots s_{l+1} s_{l+1}^{(j)}] & \text{if } j = \text{ord } s; \\ [s_1 \dots s_{l+1} push_j s_{l+1}] & \text{if } j < \text{ord } s. \end{cases} \end{aligned}$$

Convention 2.1. In the rest of the paper we will adopt the following abbreviations:

- “ $push_1^{a,j}$ ” for the operation $push_1 a^{(j,1)}$;
- “ $push_1^a$ ” for the operation $push_1 a^{(j,k)}$ where the components j and k are undetermined.

Convention 2.2 (Top stack convention). In the original definition, the operation top_i , that returns the top $(i-1)$ -stack of a higher-order stack, removes any dangling pointer resulting from the operation. Here, we suppose that top_i is

defined in such a way that all pointers are preserved. From an implementation viewpoint, since links are encoded as pairs of integers, this means that top_i just returns an unmodified copy of the top $(i - 1)$ -stack.

Definition 2.5. A tree generating *order- n collapsible pushdown automaton*, n -CPDA for short, is a tuple $\langle \Sigma, \Gamma, Q, \delta, q_0 \rangle$ where Σ is a ranked alphabet, Γ is a stack alphabet, Q is a finite set of states, q_0 is the initial state, and δ is a transition function $Q \times \Gamma \rightarrow Q \times Op + Out$ where

- Op denotes the set of operations on higher-order stacks with links from Definition 2.4.;
- $Out = \{(f, q_1, \dots, q_{arity(f)}) : f \in \Sigma, q_i \in Q\}$ denotes the set of possible output emitted at each step of the transition function.

An order- n CPDA stack is called a *configuration* of an order- n CPDA.

This definition extends the notion of tree-generating *Pushdown Automaton* (PDA) which are similarly defined on regular higher-order stack with no links and without the *collapse* stack operation.

Tree accepted by a CPDA/PDA One can think of a CPDA (resp. PDA) as a state machine equipped with a higher-order stacks. At each transition, the CPDA can either (i) applies some operation to the stack and update its state, (ii) output the root node $f \in \Sigma$ of the generated tree together with new states for each branch of the tree root. The *infinite tree accepted* by the automaton can then be obtained by recursively spawning new automata at each child of the node starting with the specified state q_i . The formal definition of the generated tree can be found in [10, 8].

Higher-order recursion schemes

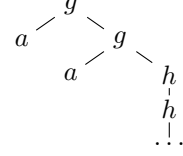
Definition 2.6. A *higher-order recursion scheme* is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$, where Σ is a ranked alphabet of *terminals*; \mathcal{N} is a finite set of typed *non-terminals*; S is a distinguished ground-type symbol of \mathcal{N} , called the start symbol; \mathcal{R} is a finite set of production rules. For each non-terminal $F : (A_1, \dots, A_n, o) \in \mathcal{N}$ there is exactly one rule of the form: $Fz_1 \dots z_m \rightarrow e$ where each z_i (called *parameter*) is a variable of type A_i and e is an applicative term of type o generated from the typed symbols in $\Sigma \cup \mathcal{N} \cup \{z_1 : A_1, \dots, z_m : A_m\}$.

We say that the recursion scheme is *order- n* just in case the order of the highest-order non-terminal is n .

Tree-generated by a recursion scheme An applicative term generated from the terminal symbols Σ only (without non-terminals), is viewed as a Σ -labelled tree and called a *value tree*. A recursion scheme defines a (potentially infinite) value tree obtained by unfolding its rewrite rules *ad infinitum*, replacing formal by actual parameters each time, starting from the start symbol S . It is formally defined as the least upper bound of the induced *schematological tree grammar* in the continuous algebra of ranked trees with the appropriate ordering [10, 7].

Example 2.1. Let G be the following order-2 recursion scheme:

$$\begin{array}{lll} S & \rightarrow & H a \\ H z & \rightarrow & F(g z) \\ F \phi & \rightarrow & \phi(\phi(F h)) \end{array}$$



with non-terminals $S : o$, $F : ((o, o), o)$, $H : (o, o)$ and terminals g, h, a of arity 2, 1, 0 respectively. Then the tree generated by G is defined by the infinite term $g a (g a (h (h (h \dots))))$ pictured on the right.

2.2 Computation tree

Informally, the *computation tree* of a higher-order recursion scheme from [11] is defined as a finite tree representation of the η -long normal form of the (possibly infinite) lambda terms inherent to the production rules of the recursion scheme.

Fix a higher-order recursion scheme $R = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$. Observe that production rules naturally map to simply-typed lambda terms by (i) currying the rule: replacing variables defined on the left-hand side by a succession of lambda abstractions on the right-hand side, (ii) interpreting terminals and non-terminal occurring in the applicative term of the right-hand side as free variables. Supposed that $F : A_1 \rightarrow \dots \rightarrow A_m \in \mathcal{N}$ for some $m \geq 0$. Then the rule $F z_1 \dots z_m \rightarrow f(G(F z_1))$ corresponds to the simply-typed lambda-term:

$$\lambda z_1 \dots z_m. f(G(F z_1))$$

of type $A_1 \rightarrow \dots \rightarrow A_m \rightarrow o$ with free variables f, F, G of appropriate type. For each non-terminal F we denote this term $\Lambda(F)$.

We recall that a simply-typed lambda term is in η -long normal form if it can be written $\lambda \bar{x}. s_0 s_1 \dots s_m$ (where $\lambda \bar{x}$ is an abbreviation for $\lambda x_1 \dots \lambda x_n$ for some $n \geq 0$) where $s_0 s_1 \dots s_m$ is of ground type, each s_j for $j \in 1..m$ is in η -long normal form, and either s_0 is a variable and $m \geq 0$; or s_0 is an abstraction $\lambda \bar{y}. s$ for some η -long normal form s and $m \geq 1$. It is convenient to write the η -long normal form of terms of ground type as $\lambda. N$ for some term N where ‘ $\lambda.$ ’ is referred to as a ‘dummy lambda’.

Observe that a purely applicative term can trivially be converted to η -long normal form by recursively η -expanding every subexpression.

We define the computation tree of a simply-typed term as an abstract syntax tree of its eta-long normal form:

Definition 2.7 (Computation tree of a simply-typed term ([2])). Let M be a simply-typed term of type T with variables names \mathcal{V} . The **computation tree** $\tau(M)$ with labels taken from $\{\text{@}\} \cup \mathcal{V} \cup \{\lambda x_1 \dots x_n \mid x_1, \dots, x_n \in \mathcal{V}, n \in \mathbb{N}\}$, is defined inductively on the η -long normal form of M as follows.

$$\begin{aligned} \tau(\lambda \bar{x}. z s_1 \dots s_m) &= \underline{\lambda \bar{x}} \langle \underline{z} \langle \tau(s_1), \dots, \tau(s_m) \rangle \rangle \\ &\quad \text{where } m \geq 0, z \in \mathcal{V}, \\ \tau(\lambda \bar{x}. (\lambda y. t) s_1 \dots s_m) &= \underline{\lambda \bar{x}} \langle \underline{\text{@}} \langle \tau(\lambda y. t), \tau(s_1), \dots, \tau(s_m) \rangle \rangle \\ &\quad \text{where } m \geq 1, y \in \mathcal{V}. \end{aligned}$$

where $l \langle t_1, \dots, t_n \rangle$ for $n \geq 0$ succinctly denotes a labelled tree with root labelled l and n ordered children trees t_1, \dots, t_n . The underlined expressions correspond to the nodes of the tree for $m > 0$, and leaves for $m = 0$.

We borrow terminology from the lambda calculus when referring to the computation tree. In particular we will sometime refer to the ‘binder’ of a variable node as the uniquely defined lambda node that binds the variable in the corresponding lambda term.

Given a simply-typed lambda term M with free variables in $\mathcal{N} \cup \Sigma$, we define the **unfolding of M** as the simply-typed lambda term M obtained by replacing every variable $N \in \mathcal{N}$ by the term $\Lambda(N)$ using capture-permitting substitution. The unfolding of the computation tree $\tau(M)$ is defined as the computation tree of the unfolding of M .

Definition 2.8. The (possibly infinite) **computation tree of a recursion scheme R** is defined as the recursive unfolding of the computation tree of the simply-typed term $\Lambda(S)$.

(An alternative definition can be found in [11].)

Remark 2.1. The repeated unfolding operation does not incur capture of variables since the lambda terms representing rewrite rules of a recursion scheme only have free variables in $\mathcal{N} \cup \Sigma$, and leave all variables in $\mathcal{N} \cup \Sigma$ unbound.

Incremental binding We recall that the order of a simple type is defined as $\text{ord}(o) = 0$ for any base type o , and $\text{ord}(A \rightarrow B) = \max(\text{ord}(A) + 1, \text{ord}(B))$. We define the order of a variable node as the order of its associated simple type, and the order of a lambda node $\lambda x_1 \cdots x_n$ for $n > 0$ as $1 + \max_{0 \leq i \leq n} \text{ord } x_i$.

We now recall the following notion from the author’s thesis [2, 4] which relates to the syntactic restriction of safety:

Definition 2.9 ([2, 4]). The computation tree of a closed lambda term is **incrementally-bound** if every variable node x is bound by the first lambda node in the path from it to the root that has order strictly greater than x .

By extension, we say that the computation tree of an open term M with free variables $x_1 \cdots x_n$, $n \geq 0$ is incrementally-bound if so is the computation tree of closed term $\lambda x_1 \cdots \lambda x_n.M$.

We extend this definition to recursion schemes through finite approximation of the computation tree: a recursion scheme is **incrementally-bound** just if every finite unfolding of the tree is incrementally-bound. (Or equivalently, if every finite tree obtained by pruning some branches of its computation tree is incrementally-bound.)

2.3 The safety restriction

The safety restriction has appeared under different forms in the literature [10, 6, 7, 2, 4]. We include here a reformulation of the definition by Knapik *et al.* in the setting of recursion schemes [10].

Definition 2.10 (Type homogeneity). We say that a simple type $A_1 \rightarrow \cdots \rightarrow A_n \rightarrow o$ with $n \geq 0$ is **homogeneous** just if $\text{ord } A_1 \geq \text{ord } A_2 \geq \cdots \geq \text{ord } A_n$, and each A_1, \dots, A_n is homogeneous [10].

Definition 2.11 (Safe recursion scheme). Let G be a higher-order recursion scheme where the *non-terminals all have homogeneous types*. We say that G is **unsafe** just if it has a production rule $Fz_1 \dots z_m \rightarrow e$ where e contains a subterm that:

1. occurs in *operand* position in e ,
2. contains a parameter of order strictly less than its order.

By “operand position” we mean “in the second position of some occurrence of the term application operator”.

A recursion scheme is said to be **safe** if it is not unsafe.

The term ‘safety’ comes from the fact that, when converted to lambda-terms, safe recursion schemes can be evaluated without ever having to generate a fresh variable during substitution: β -redexes can be reduced using (simultaneous) capture-permitting substitution [2, 4, 1].

The safe subset of the simply-typed lambda calculus introduced in [4, 2] relates to the notion of safe higher-order recursion schemes in the following sense:

Proposition 2.1 (Safe rewrite rules and lambda terms [2, Proposition 3.11]). *Take a recursion scheme $R = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$. For every non-terminal N , the associated rewriting rule is safe if and only if the simply-type term $\Lambda(N)$, with free variables in $\Sigma \cup \mathcal{N}$, is derivable with the typing judgment of the safe lambda calculus of [2] and all types involved in the typing derivation are homogeneous.*

It was shown that incremental-binding characterizes safe simply-typed lambda terms [2, Proposition 5.11]. We show here the equivalent result for recursion schemes. Without loss of generality we will consider recursion schemes **with no dead rule** such that for every production rule, there exists a derivation from the start non-terminal involving that rewrite rule.

Proposition 2.2 (Binder characterization for HORS). *A higher-order recursion scheme with no dead rule is safe (in the sense of [10]) if and only if it is homogeneous and incrementally-bound.*

Proof. The proof reduces to the setting of the lambda calculus where a similar result was shown for finite lambda terms [2]. Take a safe recursion scheme $R = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$.

(\Rightarrow) Observe that all the iterated unfoldings of $\Lambda(S)$ are safe homogeneous simply-typed terms. It’s true of $\Lambda(S)$ itself by Proposition 2.1. And by the Substitution Lemma [2, 3.19], it is also true of each subsequent unfolding. Consequently, by the characterization result of the safe-simply typed lambda calculus [2, Proposition 5.11](i), every repeated unfolding of $\Lambda(S)$ has an incrementally-bound computation tree. Hence R is incrementally-bound.

(\Leftarrow) Assume that R is not safe. Suppose that it’s homogeneously-typed, then the syntactic constraint of safety is not met for some production rule $Nx_1 \cdots x_n \rightarrow e \in \mathcal{R}$ and non-terminal $N \in \mathcal{N}$. Since the scheme has no dead-rule, after performing $|\mathcal{N}|$ unfoldings of $\Lambda(S)$ the non-terminal N must have been substituted at least once by $\Lambda(N) = \lambda x_1 \cdots x_n. e$, which is unsafe by Proposition 2.1. Hence after a finite number of unfoldings of S we obtain a term t' containing the unsafe subterm $\Lambda(N)$. Thus, by [2, Proposition 5.11](ii), the computation tree t' is not incrementally-bound. \square

2.4 Equi-expressivity results

Now that we have defined the notions of tree-generating higher-order recursion schemes and tree-generating higher-order automata we can state two important results about their relative expressivity:

Theorem 2.1 (Safe HORS - PDA equi-expressivity [10]). *A Σ -labelled tree is generated by a safe order- n recursion scheme if and only if it is accepted by an order- n pushdown automaton.*

Theorem 2.2 (HORS - CPDA equi-expressivity [9]). *A Σ -labelled tree is generated by an order- n recursion scheme if and only if it is accepted by an order- n collapsible pushdown automaton.*

The proof of the HORS-CPDA equi-expressivity result is constructive in both direction: given a higher-order recursion scheme R , one can construct a collapsible pushdown automaton denoted $CPDA(G)$ that recognizes the same tree; and given a collapsible pushdown automaton A one can construct a higher-order recursion scheme recognizing the same tree. In what follows, we will refer to them as the HMOS constructions.

The following result summarizes the contribution of the present paper:

Theorem 2.3 (Homogeneity is not a restriction). *A Σ -labelled tree is generated by an incrementally-bound order- n recursion scheme if and only if it is accepted by an order- n pushdown automaton.*

Section 4 gives a constructive proof of the implication by showing that the collapsible pushdown automaton $CPDA(G)$ from the HMOS construction can be turned into an equivalent pushdown automaton (Theorem 4.1). Section 5 proves the opposite direction.

3 From recursion scheme to collapsible pushdown automaton

In this section we recall some of the concepts from the HMOS constructions from Theorem 2.2. Familiarity with the original construction can be helpful, in particular for the concept of traversals. We refer the reader to [9] for a more detailed introduction to those concepts.

We fix a higher-order recursion scheme $G = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$ of order n . Let N denotes the set of nodes of the computation tree of G ; $N_{@}$ denotes the set of application nodes; N_{λ} the set of lambda nodes; $N_{\lambda}^{\text{prime}}$ the set of lambda nodes that are the first child of some @ nodes (also known as *prime* lambda nodes); and N_{var} denotes the set of variables nodes.

Presentation

Definition 3.1 (CPDA of G). We consider the order n collapsible pushdown automaton, obtained from G by the HMOS transformation defined in [9] and denoted $CPDA(G)$ [8, Definition 5.2]. Recall that:

- The ranked-alphabet is Σ (the alphabet of G);

- The stack-alphabet Γ of the automaton is taken as the set of nodes of the computation *graph* of G . Alternatively, this graph can be viewed as the computation tree of $\Lambda(S)$ by replacing back-pointers in the computation graph with pointers to non-terminals nodes in the computation tree. Thus $\Gamma = N$;
- The transition function δ of the automaton is shown in Figure 1 using a more concise definition than the original one from [8];
- The initial configuration is defined as $c_0 = \text{push}_1 \lambda \perp_n$ where λ refers to the root (dummy) lambda node of the computation graph of G .

The automaton is well-defined in the sense that no collapse can occur at an element whose link is undefined: In particular *collapse* never occurs at non-lambda nodes. It is equivalent to G in the sense that they both generate the same tree [9]. The idea is that automaton $CPDA(G)$ proceeds by inductively computing a set of traversals over the computation tree of G . The traversals are shown to correctly evaluate the production rules of the recursion scheme G , therefore the constructed automaton correctly accepts the same tree as the recursion scheme.

(At this point, readers not familiar with the concept of traversals over the computation graph of a recursion scheme, O-view and P-view may want to lookup their definitions in [11].)

Observe that one can easily modify $CPDA(G)$ into an automaton that “prints out” the traversal that is being computed. This can be done by changing the behaviour of the push_1 operation to make it print out the input element before pushing it on the stack. The justification pointers can then be recovered inductively using the node labels: For a variable node, it is the only node-occurrence that binds it in the P-view at that point (which is computable by the induction hypothesis); prime lambda nodes always point to their immediate predecessor; and a non-prime lambda node $\lambda \bar{x}$ is always justified by the predecessor of the justifier of the variable node preceding it (written $\text{ip}(\text{jp}(t))$ where t is the traversal ending with $\lambda \bar{x}$).

Remark 3.1. Our presentation of δ differs slightly from the original one: In the case (A), when pushing the prime child of an application node $@$ on the stack, we assign it a link pointing to the preceding stack symbol in the top 1-stack (*i.e.*, the $@$ -node itself). This modification avoids the case analysis on the value of j —the child-index of u ’s binder—in the cases (V_0) and (V_1) , and the sequence of instructions pop_1^{p+1} can just be replaced by pop_1^p ; *collapse*.

The stack of the current configuration alone does not suffice to reconstruct the traversal that is being computed due to the use of the “destructive” operations *collapse* in the CPDA. Nevertheless, two important pieces of information are recoverable from the configuration-stack: the O-view and the P-view of the traversal.

Proposition 3.1 (Recovering views from a configuration of $CPDA(G)$). *Let c be a configuration of $CPDA(G)$. The **long O-view**, **O-view** and **P-view** of the traversal currently being simulated by the configuration c , written respectively $\lfloor c \rfloor$, $\lrcorner c \lrcorner$ and $\lceil c \rceil$, can be retrieved using the following stack operations:*

If u 's label is not a variables, the action is just a $push_1^v$, where v is an appropriate child of the node u . Precisely:

- (A) If the label is an @ then $\delta(u) = push_1^{E_0(u),1}$.
- (S) If the label is a Σ -symbol f then $\delta(u) = push_1^{E_i(u)}$, where $1 \leq i \leq ar(f)$ is the direction requested by the Environment, or Opponent.
- (L) If the label is a lambda then $\delta(u) = push_1^{E_1(u)}$.

Suppose u is a variable which is the i -parameter of its binder and let p be the span of u .

- (V_1) If the variable has order $l \geq 1$, then

$$\delta(u) = push_{n-l+1}; pop_1^p; collapse; push_1^{E_i(top_1), n-l+1}$$

- (V_0) If the variable is of ground type then

$$\delta(u) = pop_1^p; collapse; push_1^{E_i(top_1)}$$

Figure 1: The transition rules of $CPDA(G)$.

- *Long O-view:*

$$\lfloor s \rfloor = \begin{cases} \epsilon & \text{if } top_1 s \text{ is undefined;} \\ \lfloor pop_1 s \rfloor \cdot top_1 s & \text{if } top_1 s \in N_{\text{var}}, \text{ } top_1 s \text{ pointing to its immediate predecessor;} \\ \lfloor pop_1 s \rfloor \cdot top_1 s & \text{if } top_1 s \in N_{@}, @ \text{ having no pointer;} \\ \lfloor collapse s \rfloor \cdot top_1 s & \text{if } top_1 s \in N_{\lambda}^{\text{prime}}, \text{ } top_1 s \text{ pointing to its immediate predecessor;} \\ \lfloor collapse s \rfloor \cdot top_1 s & \text{if } top_1 s \in N_{\lambda} \setminus N_{\lambda}^{\text{prime}}, \text{ } top_1 s \text{ pointing to } ip(jp(s)). \end{cases}$$

- The *O-view* is defined similarly to the long-*O-view* except that the calculation stops when an @-node is reached:

$$\lfloor s \rfloor = top_1 s \quad \text{if } top_1 s \in N_{@}, @ \text{ having no pointer;}$$

- As shown in the HMOS construction, the *P-view* $\lceil s \rceil$ is given by $top_2 c$ [8].

Proof. This follows from the inductive definition of traversals of $CPDA(G)$, *P*-views and *O*-views [8]. \square

Reachable configurations We recall the notion of reachability with respect to the \rightarrow -step relation as defined in [8]: for two configurations c and c' we have $c \rightarrow c'$ just if $c' = \delta(\text{top}_1 c)(c)$ where δ is the transition function of the $CPDA(G)$. In other words, a configuration is \rightarrow -**reachable** if it can be attained starting from the initial configuration c_0 by performing one or more applications of the steps (A), (S), (L), (V_1) , (V_0) from the algorithm defining $CPDA(G)$.

The intermediate configurations visited by the internal transitions within a step are therefore not \rightarrow -reachable. A configuration is said to be **reachable** if it is \rightarrow -reachable or if it is an intermediate configuration computed during a \rightarrow -step (*i.e.*, if it can be written $(op_1; \dots; op_k)(c)$ where c is \rightarrow -reachable and op_1, \dots, op_k are the first k instructions of some \rightarrow -step).

Link convention Observe that in $CPDA(G)$, when we push a lambda node $\lambda \bar{\xi}$ on the stack, the associated link has order 1 if it is a prime lambda node (case (A)), and $n - \text{ord } \lambda \bar{\xi} + 1$ otherwise (case (V_1)). Hence, since no CPDA instruction can change the link order of an element pushed on the stack, at every stage during the execution of the CPDA the link order can be recovered from the (order of the) node itself.

From now on we will only work with stacks occurring as sub-stack of reachable configurations of $CPDA(G)$ therefore we will omit the order-component altogether when representing stack symbols: we write $\lambda \bar{\xi}^k$ to mean $\lambda \bar{\xi}^{(1,k)}$ if $\lambda \bar{\xi}$ is a prime lambda node and $\lambda \bar{\xi}^{(n - \text{ord } \lambda \bar{\xi} + 1, k)}$ otherwise.

4 From incrementally-bound recursion schemes to pushdown automata

We now fix an **incrementally-bound** higher-order recursion scheme G of order n . We first give a detailed analysis of $CPDA(G)$ and then show how to derive an equivalent (non-collapsible) order- n pushdown automaton.

4.1 Incremental order-decomposition

Observation Let s be a 1-stack. For any $l \in \mathbb{N}$, s can then be written

$$s = u_{r+1} \cdot \lambda \bar{\eta}_r^{k_r} \cdot u_r \cdot \dots \cdot \lambda \bar{\eta}_1^{k_1} \cdot u_1$$

where

- $\lambda \bar{\eta}_1^{k_1}$ is the last λ -node in s with order strictly greater than l ;
- for $1 < l \leq r$, $\lambda \bar{\eta}_l^{k_l}$ is the last λ -node in $s_{\leq \lambda \bar{\eta}_{l-1}^{k_{l-1}}}$ with order strictly greater than $\text{ord } \lambda \bar{\eta}_{l-1}^{k_{l-1}}$,
- r is defined as the smallest number such that $s_{\leq \lambda \bar{\eta}_r^{k_r}}$ does not contain any lambda node of order strictly greater than $\lambda \bar{\eta}_r^{k_r}$.

In other words:

- for $1 \leq k \leq r$, all the lambda nodes occurring in u_l have order strictly smaller than $\text{ord } \lambda \bar{\eta}_l$;

- for $1 \leq l < l' \leq r$ we have $\text{ord } \lambda \overline{\eta}_l^{k_l} < \text{ord } \lambda \overline{\eta}_{l'}^{k_{l'}}$;
- $r = 0$ if and only if all the lambda node in s have order $\geq l$.

The subsequence $\lambda \overline{\eta}_r^{k_r} \dots \lambda \overline{\eta}_1^{k_1}$ of s consisting of the lambda nodes $\lambda \overline{\eta}_l^{k_l}$ defined above is called the **incremental order-decomposition of the 1-stack s with respect to $l \in \mathbb{N}$** . This sequence is uniquely determined for any given $l \in \mathbb{N}$.

We now generalize this notion to higher-order stacks.

Definition 4.1. The **incremental order-decomposition of a higher-order stack s with respect to $l \in \mathbb{N}$** (or order-decomposition for short), written $\text{orddec}_l(s)$, is defined as the order-decomposition of the top order-1 stack (defined using convention 2.2). Equivalently it can be defined as follows:

$$\begin{aligned} \text{orddec}_l(\epsilon) &= \epsilon \\ \text{for } s \neq \epsilon \quad \text{orddec}_l(s) &= \text{orddec}_{\text{ord } \lambda \overline{\eta}}(s_{< \lambda \overline{\eta}}) \cdot \lambda \overline{\eta}^k \\ &\quad \text{where } \lambda \overline{\eta}^k \text{ is the last node in } \text{top}_2 s \text{ with order } > l. \end{aligned}$$

The **incremental order-decomposition** of s , written $\text{orddec}(s)$, is defined as:

$$\text{orddec}(s) = \text{orddec}_0(s) .$$

It follows immediately from the definition that:

$$l < l' \implies \text{orddec}_{l'}(s) \leq \text{orddec}_l(s) \quad (1)$$

where \leq denotes the sequence-prefix ordering.

Lemma 4.1. Let s be a (possibly higher-order) stack such that $\text{top}_1 s \in N_{\text{at}} \cup N_{\text{var}}$ and $l \geq 0$.

- (i) Suppose that $\text{orddec}_l(s) = \langle \lambda \overline{\eta}_r^{k_r}, \dots, \lambda \overline{\eta}_1^{k_1} \rangle$ then for any lambda node $a \in \Gamma$ and link (j, k) we have

$$\text{orddec}_l(\text{push}_1 a^{(j,k)} s) = \begin{cases} \text{orddec}_l(s), & \text{if } \text{ord } a \leq l; \\ \langle a^k \rangle, & \text{if } \text{ord } a \geq \text{ord } \lambda \overline{\eta}_r; \\ \langle \lambda \overline{\eta}_r^{k_r}, \dots, \lambda \overline{\eta}_i^{k_i}, a^k \rangle, & \text{otherwise,} \\ & i = \min\{i \in \{1..r\} \mid \text{ord } a < \text{ord } \lambda \overline{\eta}_i\} . \end{cases}$$

- (ii) For any non-lambda node $a \in \Gamma$ and link (j, k) we have

$$\text{orddec}_l(\text{push}_1 a^{(j,k)} s) = \text{orddec}_l(s) .$$

- (iii) If $\text{top}_1 s$ is not a lambda node then

$$\text{orddec}_l(\text{pop}_1 s) = \text{orddec}_l(s) .$$

Proof. Follows immediately from the definition of $\text{orddec}_l(s)$. \square

Lemma 4.2 (Incremental binders are in the order-decomposition). Let c be a \rightarrow -reachable configuration of $\text{CPDA}(G)$ such that $\text{top}_1 c$ is a variable x . Then

- (i) $\text{orddec}(c)$ contains at least a node with order strictly greater than $\text{ord}(x)$;
- (ii) the last lambda node in $\text{orddec}(c)$ satisfying the first condition is precisely x 's binder.

In other words, x 's binder is the last lambda node in $\text{orddec}_{\text{ord } x}(c)$.

Proof. (i) The top 1-stack of a \rightarrow -reachable configuration contains the P-view of some traversal whose last visited node is the top_1 symbol [8, Corollary 8]; and the P-view of a traversal is exactly the path (in the unfolding of) the computation graph from the last visited node to the root [11, Proposition 6]. Hence the binder of x , whose order is strictly greater than $\text{ord } x$, occurs in the top 1-stack. Consequently $\text{orddec}(c)$ must contain at least one node of order strictly greater than l .

- (ii) Since the recursion scheme is incrementally-bound, x 's binder is precisely the first λ -node in the path to the root in the computation tree with order strictly greater than x . \square

4.2 Stack safety

Definition 4.2 (Safe stack). Let s be an order- j non-empty stack for $j \geq 1$.

The stack s is *l -safe* iff

1. $\text{orddec}_l(s) = \langle \lambda \bar{\eta}_r^1, \dots, \lambda \bar{\eta}_1^1 \rangle$ for some $r \geq 0$ i.e., the height component of the links in $\text{orddec}_l(s)$ are all equal to 1;
2. for all $1 \leq q \leq r$ such that $n - \text{ord } \lambda \bar{\eta}_q + 1 \leq \text{ord } s$:
 - for $q = 1$ we have $\text{collapse } s_{\leq \lambda \bar{\eta}_1}$ is l -safe;
 - for $q > 1$ we have $\text{collapse } s_{\leq \lambda \bar{\eta}_q}$ is $\text{ord}(\lambda \bar{\eta}_{q-1})$ -safe.

We say that s is *safe* if it is 0-safe.

Since s is a stack, and not necessarily a configuration, it may have dangling pointers. The condition $n - \text{ord } \lambda \bar{\eta}_q + 1 \leq \text{ord } s$ in the definition ensures that $\lambda \bar{\eta}_j$'s link is not dangling so that we can indeed perform a collapse at $\lambda \bar{\eta}_j$.

Lemma 4.3. Let s be a stack such that $\text{orddec}_l(s) = \langle \lambda \bar{\eta}_r^{k_r}, \dots, \lambda \bar{\eta}_1^{k_1} \rangle$ and $l \geq 0$. If s is l -safe and $l \leq k \leq n$ then s is k -safe.

Proof. Immediate consequence of the definition. \square

Lemma 4.4 (Collapse simulation). Let s be a sub-stack of a reachable configuration of $\text{CPDA}(G)$ and $l \geq 0$. If $\text{ord } s \geq 2$ and $\text{top}_2 s$ is l -safe or if $\text{ord } s = 1$ and s is l -safe then for any lambda node $\lambda \bar{\eta}$ in $\text{orddec}_l(s)$ we have:

$$\text{collapse } s_{\leq \lambda \bar{\eta}} = \begin{cases} \text{pop}_1 s_{\leq \lambda \bar{\eta}} & \text{if } \lambda \bar{\eta} \text{ is prime,} \\ \text{pop}_{n - \text{ord } \lambda \bar{\eta} + 1} s_{\leq \lambda \bar{\eta}} & \text{otherwise.} \end{cases}$$

Proof. The collapse operation is defined as $\text{collapse } s = \text{pop}_j^k s$ where $(j, k) \in \mathbb{N} \times \mathbb{N}$ is the link attached to $\text{top}_1 s$. Since s is a sub-stack of a reachable configuration we have $j = 1$ if $\lambda \bar{\eta}$ is prime and $j = n - \text{ord } \lambda \bar{\eta} + 1$ otherwise. Furthermore, since $\text{top}_2 s$ is safe and $\lambda \bar{\eta}$ belongs to the order-decomposition, the height component necessarily equals 1. \square

4.2.1 Operations preserving stack safety

Lemma 4.5. *Let s be a higher-order stack. Suppose that s is l -safe, $l \geq 0$. Then:*

- (i) *$\text{top}_{\text{ord } s} s$ is l -safe;*
- (ii) *if $\text{top}_1 s$ is not a lambda node then $\text{pop}_1 s$ is l -safe;*
- (iii) *for every non-lambda node a , $1 \leq j \leq n$, $k \geq 1$, $\text{push}_1 a^{(j,k)} s$ is l -safe;*
- (iv) *for every lambda node a , $\text{push}_1 a^{(1,1)} s$ is l -safe if $\text{ord } a < l$, and safe if $\text{ord } a \geq l$.*

Proof. This is a direct consequence of Lemma 4.1. For (vi), the cases $\text{ord } a > l$ and $\text{ord } a < l$ follow immediately from Lemma 4.1(i); for $\text{ord } a = l$ it follows from the fact that $\text{orddec}_0(\text{push}_1 a^{(1,1)} s) = \text{orddec}_l(s) \cdot a^1$. \square

Lemma 4.6. *Let $0 \leq l < n$, $q \geq 0$ and s be a stack of level $1 \leq \text{ord } s < n$. If s is q -safe then $s^{(n-l+1)}$ is $\max(l, q)$ -safe.*

Proof. Let s be a safe stack with $1 \leq \text{ord } s < n$. We prove the result by induction on the size of s . The base case is the trivial: s is the empty stack. Step case: Since s is q -safe we have $\text{orddec}_q(s) = \langle \lambda \bar{\eta}_r^1, \dots, \lambda \bar{\eta}_1^1 \rangle$. By (1), $\text{orddec}_{\max(l, q)}(s)$ is a prefix of $\text{orddec}_q(s)$. Let b be the index in $\text{orddec}_q(s)$ of the last node of $\text{orddec}_{\max(l, q)}(s)$: thus $\lambda \bar{\eta}_b^1$ is the last lambda node in $\text{top}_2 s$ with order $> \max(l, q)$.

The stack-operation $(\cdot)^{(n-l+1)}$ updates the pointers as follows: the height component of each link is incremented if the order of the stack symbol is l and is kept unchanged otherwise. Hence we have:

$$\text{orddec}_q(s^{(n-l+1)}) = \langle \lambda \bar{\eta}_r^1, \dots, \lambda \bar{\eta}_b^1, \lambda \bar{\eta}_{b-1}^k, \lambda \bar{\eta}_{b-2}^1, \dots, \lambda \bar{\eta}_1^1 \rangle$$

for some $1 \leq k \leq 2$. And therefore:

$$\text{orddec}_{\max(l, q)}(s^{(n-l+1)}) = \langle \lambda \bar{\eta}_r^1, \dots, \lambda \bar{\eta}_b^1 \rangle. \quad (2)$$

Now consider an index j such that $b \leq j \leq r$ and $n - \text{ord } \lambda \bar{\eta}_j + 1 \leq \text{ord } s$. Since the height component of $\lambda \bar{\eta}_j$'s link is not affected by the operation $(\cdot)^{(n-l+1)}$, this operation commutes with *collapse* and we have:

$$\text{collapse } s_{\leq \lambda \bar{\eta}_j}^{(n-l+1)} = (\text{collapse } s_{\leq \lambda \bar{\eta}_j})^{(n-l+1)}. \quad (3)$$

By assumption s is q -safe therefore $\text{collapse } s_{\leq \lambda \bar{\eta}_j}$ is q -safe if $j = b = 1$ and $\text{ord}(\lambda \bar{\eta}_{j-1})$ -safe if $j > b \geq 1$.

Since $\text{collapse } s_{\leq \lambda \bar{\eta}_j}$ is strictly smaller than s , by the induction hypothesis we have that $(\text{collapse } s_{\leq \lambda \bar{\eta}_j})^{(n-l+1)}$ is $\max(q, l)$ -safe if $j = b = 1$, and $\max(\text{ord}(\lambda \bar{\eta}_{j-1}), l)$ -safe if $j \geq b > 1$.

For $j > b$, $\lambda \bar{\eta}_{j-1}$ occurs in $\text{orddec}_l s$ therefore $\text{ord}(\lambda \bar{\eta}_{j-1}) > l$, similarly for $j = b$ we have $\text{ord}(\lambda \bar{\eta}_{j-1}) \leq l$. Hence $(\text{collapse } s_{\leq \lambda \bar{\eta}_j})^{(n-l+1)}$ is

- (i) $\max(q, l)$ -safe for $j = b = 1$,
- (ii) l -safe for $j = b > 1$, and therefore $\max(q, l)$ -safe by Lemma 4.3,

(iii) $\lambda\bar{\eta}_{j-1}$ -safe for $j > b$.

This shows that $(collapse\ s_{\leq \lambda\bar{\eta}_j})^{(n-l+1)}$ is $\max(l, q)$ -safe, and therefore by (3) so is $collapse\ s_{\leq \lambda\bar{\eta}_j}^{(n-l+1)}$. \square

Lemma 4.7. *Let s be a higher-order stack of level ≥ 2 and $l \geq 0$. If*

1. $pop_{ord\ s}\ s$ is safe,
2. and $top_{ord\ s}\ s$ is l -safe,

then s is l -safe.

Proof. Let $s = [s_1 \dots s_r\ s_{r+1}]$ for some $l \geq 0$. We proceed by induction on $top_{ord\ s}\ s = s_{r+1}$. The base case $s_{r+1} = \perp_{ord\ s-1}$ is trivial. Suppose that s_{r+1} is not the empty stack.

- (i) Clearly $orddec_l\ s = orddec_l\ s_{r+1}$, hence since s_{r+1} is l -safe the lambda nodes in $orddec_l\ s$ have all a link of height 1.
- (ii) Let $\lambda\bar{\eta}$ be a lambda node in $orddec_l(s) = orddec_l(s_{r+1})$ such that $n - ord\ \lambda\bar{\eta} + 1 \leq ord\ s$. Since its link is of height 1 we have $collapse\ s_{\leq \lambda\bar{\eta}} = pop_{n-ord\ \lambda\bar{\eta}+1}\ s_{\leq \lambda\bar{\eta}}$.

If $n - ord\ \lambda\bar{\eta} + 1 = ord\ s$ then $pop_{n-ord\ \lambda\bar{\eta}+1}\ s_{\leq \lambda\bar{\eta}} = pop_{ord\ s}\ s_{\leq \lambda\bar{\eta}} = pop_{ord\ s}\ s$ which is l -safe by the first assumption and Lemma 4.3.

Otherwise $n - ord\ \lambda\bar{\eta} + 1 < ord\ s$ and we have:

$$\begin{aligned} collapse\ s_{\leq \lambda\bar{\eta}} &= pop_{n-ord\ \lambda\bar{\eta}+1}\ s_{\leq \lambda\bar{\eta}} && \text{since } \lambda\bar{\eta}'\text{'s link has height 1} \\ &= [s_1 \dots s_l\ (pop_{n-ord\ \lambda\bar{\eta}+1}\ s_{r+1 \leq \lambda\bar{\eta}})] && n - ord\ \lambda\bar{\eta} + 1 < ord\ s \\ &= [s_1 \dots s_p\ (collapse\ s_{r+1 \leq \lambda\bar{\eta}})] && \text{since } \lambda\bar{\eta}'\text{'s link has height 1.} \end{aligned}$$

By the second assumption, s_{r+1} is l -safe therefore $collapse\ s_{r+1 \leq \lambda\bar{\eta}}$ is l -safe if $\lambda\bar{\eta}$ is the last node in the l -order decomposition, and k -safe where k is the order of the following node in $orddec_l(s_{r+1})$ otherwise.

Since $|collapse\ s_{r+1 \leq \lambda\bar{\eta}}| < |s_{r+1}|$ we can use the induction hypothesis to show that the same hold for $[s_1 \dots s_p\ (collapse\ s_{r+1 \leq \lambda\bar{\eta}})]$. Therefore it is l -safe and so is $collapse\ s_{\leq \lambda\bar{\eta}}$ by the previous equality. \square

Lemma 4.8. *Let $n > l \geq 1$ and s be a safe higher-order stack such that $2 \leq n - l + 1 \leq ord\ s \leq n$. Then $push_{n-l+1}\ s$ is l -safe.*

Proof. Let $s = [s_1 \dots s_{c+1}]$ be a safe higher-order stack such that $2 \leq n - l + 1 \leq ord\ s \leq n$. Then by Lemma 4.5(i), s_{c+1} is safe.

We show that $push_{n-l+1}\ s$ is l -safe by finite induction on the order of s .

- Base case: $ord\ s = n - l + 1$. We have $push_{n-l+1}\ s = [s_1 \dots s_{c+1} s_{c+1}^{(n-l+1)}]$. Since s_{c+1} is safe, by Lemma 4.6 $s_{c+1}^{(n-l+1)}$ is l -safe, and by Lemma 4.7, $[s_1 \dots s_{c+1} s_{c+1}^{(n-l+1)}]$ is l -safe.
- Suppose $ord\ s > n - l + 1$. Then $push_{n-l+1}\ s = [s_1 \dots s_{c+1} push_{n-l+1}\ s_{c+1}]$. Since s_{c+1} is safe, by the induction hypothesis $push_{n-l+1}\ s_{c+1}$ is l -safe, and by Lemma 4.7 so is $[s_1 \dots s_{c+1} push_{n-l+1}\ s_{c+1}]$. \square

4.3 Simulation and proof of correctness

Proposition 4.1. *Let G be an incrementally-bound recursion scheme. The \rightarrow -reachable configurations of $CPDA(G)$ are safe.*

Proof. If $n = \text{ord } c = 1$ then the result holds trivially since $CPDA(G)$ does not contain any transition of the form $push_j$ for $j > 1$ and therefore the links in a reachable configuration all have a height component equal to 1.

Take $n \geq 2$. We proceed by induction on the \rightarrow -step relation. The initial configuration is clearly safe. Suppose that c is a safe \rightarrow -reachable configuration and that $c \rightarrow c'$. We do a case analysis on $top_1 c$:

- (A): We have $c' = push_1^{E_0(u),1} c = push_1 E_0(u)^{(1,1)} c$ where $E_0(u)$ denotes a lambda node. It is safe by Lemma 4.5(iv).
- (S): We have $c' = push_1^a c = push_1 a^{(j,k)} c$ for some dummy lambda node a and undetermined link (j, k) . It is safe by Lemma 4.5(iv) since $\text{ord } a = 0$.
- (L): We have $c' = push_1^{E_1(u)} c = push_1 E_1(u)^{(j,k)}$ where $E_1(u)$ is not a lambda node and $j, k \geq 1$ are undetermined. It is safe by Lemma 4.5(iii).
- $(V_1) \ \& \ (V_0)$: Suppose u is labelled by a variable x of order l . Since c is safe we have $\text{orddec}(c) = \langle \lambda \bar{\eta}_r^1, \dots, \lambda \bar{\eta}_1^1 \rangle$, $r \geq 0$. Since the recursion-scheme G is safe, by Lemma 4.2, x 's binder is precisely the last node of $\text{orddec}_l(c)$. Let b be its index in $\text{orddec}(c)$, and $i \geq 1$ be the index of x in $\bar{\eta}_b$.
 - (V_1) : $l \geq 1$. We have $c' = push_1 E_i(top_1)^{(n-l+1,1)} t$ where t is given by $(push_{n-l+1}; pop_1^p; collapse)(c) = collapse((push_{n-l+1} c)_{\leq \lambda \bar{\eta}_b})$. By Lemma 4.8 $push_{n-l+1} c$ is l -safe therefore, since $\lambda \bar{\eta}_b$ is the last node in $\text{orddec}_l(c)$, by definition of l -safety we have that t is l -safe. Finally the lambda node $E_i(top_1)$ pushed by the last operation has precisely order $l = \text{ord}(x)$ therefore

$$\text{orddec}_0(c') = \langle \lambda \bar{\eta}_r^1, \dots, \lambda \bar{\eta}_b^1, E_i(top_1(t))^1 \rangle.$$

Thus all the lambda nodes in $\text{orddec}_0(c')$ have a link of height 1.

We now need to show that safety is preserved when collapsing at nodes of $\text{orddec}_0(c')$. Let $b \leq j \leq r$, we have $c'_{\leq \lambda \bar{\eta}_j} = t_{\leq \lambda \bar{\eta}_j}$. For $j > b$, the l -safety of t implies that $collapse c'_{\leq \lambda \bar{\eta}_j}$ is $\text{ord } \lambda \bar{\eta}_{j-1}$ -safe as required. For $j = b$ it gives that $collapse c'_{\leq \lambda \bar{\eta}_b}$ is l -safe as required since $l = \text{ord } E_i(top_1(t))$.

Now it remains to show that $collapse(c'_{\leq E_i(top_1(t))}) = collapse c'$ is safe. Since we have $i \geq 1$ the node $top_1(c') = E_i(top_1(t))$ is not a prime lambda node, thus by Lemma 4.4 we can simulate the $collapse$ by a pop of order $n - \text{ord } E_i(top_1(t)) + 1$:

$$\begin{aligned} collapse c' &= pop_{n - \text{ord } E_i(top_1(t)) + 1} c' \\ &= pop_{n-l+1} c' \\ &= (push_{n-l+1}; pop_1^p; collapse; push_1^{E_i(top_1), n-l+1}; pop_{n-l+1}) c \end{aligned}$$

The operation pop_1 and $push_1$ only affects the top 1-stack. Furthermore, since x 's binder has order $> l$, its link has order $< n - l + 1$

therefore the collapse operation following the pop_1^p only affects the top $(n-l)$ -stack. Consequently, the operation pop_{n-l+1} effectively restores the configuration to its value prior to performing the $push_{n-l+1}$ operation:

$$collapse\ c' = c .$$

Hence c' is safe.

- (V_0) : $l = 0$ (which implies that $b = 1$). The configuration c' is given by $push_1 E_i(top_1) collapse(c_{\leq \lambda \overline{\eta}_b})$. Since c is safe by definition so is $collapse(c_{\leq \lambda \overline{\eta}_b})$. Since the pushed lambda node $E_i(top_1)$ has order $l = 0$, by Lemma 4.5(iv) c' is safe. \square

Definition 4.3 (Simulating PDA). Let G be an incrementally-bound recursion scheme. We define $PDA(G)$ as the higher-order PDA obtained from $CPDA(G)$ by replacing every $collapse$ operation by pop_1 if $top_1 s$ is prime, and by $pop_{n-\text{ord } top_1(s)+1}$ otherwise.

Theorem 4.1 (Correctness of the simulation). *$PDA(G)$ and $CPDA(G)$ are equivalent.*

Proof. In $CPDA(G)$, the collapse operation occurs only in the steps (V_1) and (V_0) . For (V_1) it is of the form:

$$collapse(pop_1^p(push_{n-l+1} c))$$

for some \rightarrow -reachable configuration c , where $top_1 c$ is a variable x of order l and span p . By the previous proposition, c is safe and by Lemma 4.8 $push_{n-l+1} c$ is l -safe. Since x has span p , after the operation pop_1^p the top stack symbol is precisely x 's binder, which by Lemma 4.2, belongs to $\text{orddec}_l(c)$, therefore by Lemma 4.4 the collapse can be soundly simulated by a pop of order 1 if x 's binder is a prime node, and a pop of order $n - \text{ord}(top_1(pop_1^p(push_{n-l+1} c))) + 1$ otherwise.

The case (V_0) is proved similarly. \square

Remark 4.1. The result also holds for the slightly different definition of the automaton $CPDA(G)$ from the original HMOS transformation [9]: Clearly the two CPDAs have the same set of \rightarrow -reachable configurations so Proposition 4.1 clearly still holds. The simulating PDA from Def. 4.3, however, is obtained by replacing every $collapse$ operation in the transition of the CPDA by $pop_{n-\text{ord } top_1(s)+1}$. Also the equality in Lemma 4.4 becomes:

$$collapse\ s_{\leq \lambda \overline{\eta}} = pop_{n-\text{ord } \lambda \overline{\eta}+1}\ s_{\leq \lambda \overline{\eta}} .$$

The correctness of the simulation follows similarly.

5 From PDA to incrementally-bound RS

Proposition 5.1. *Let \mathcal{A} be an order- n PDA, and Σ a ranked alphabet. There exists an order- n incrementally-bound recursion scheme R such that for every Σ -labelled tree t , t is accepted by \mathcal{A} if and only if it is generated by R .*

Proof. Let \mathcal{A} be an order- n PDA. Following the HMOS transformation from order- n CPDA to order- n recursion scheme [9] we show how to produce an equivalent homogeneously-typed recursion scheme that is incrementally-bound.

Let's consider the PDA \mathcal{A} as a CPDA. The HMOS transformation yields an equivalent recursion scheme $R = \langle \Sigma, \mathcal{N}, \mathcal{R}, S \rangle$. (Note: we refer the reader to [9] for the definition of the production rules \mathcal{R} .)

Since a PDA, viewed as a CPDA, never makes use of the *collapse* operation, we can get rid of the parameters $\overline{\Phi}$ from the HMOS construction, used to implement the collapse stack operation via production rules. With this simplification, the type of the non-terminal $\mathcal{F}_p^{a,e}$ for each stack symbol a , $1 \leq e \leq n$, and state $1 \leq p \leq m$ becomes:

$$\mathcal{F}_p^{a,e} : (n-1)^m \rightarrow \dots \rightarrow 0^m \rightarrow 0$$

which is homogeneously-typed.

The production rules get simplified to the general form:

$$\mathcal{F}_p^{a,e} \overline{\Psi_{n-1}} \dots \overline{\Psi_0} \xrightarrow{(q,\theta)} \Xi_{(q,\theta)}$$

for every state q and stack operation θ of the PDA's transition function, where the right-hand side $\Xi_{(q,\theta)}$ is given by the following table:

Operation θ	Applicative term $\Xi_{(q,\theta)}$
$push_1^{b,k}$	$\mathcal{F}_q^{b,k} \langle \mathcal{F}_i^{a,e} \overline{\Psi_{n-1}} i \rangle \overline{\Psi_{n-2}} \dots \overline{\Psi_0}$
$push_j$	$\mathcal{F}_q^{a,e} \overline{\Psi_{n-1}} \dots \overline{\Psi_{n-(j-1)}} \langle \mathcal{F}_i^{a,e} \overline{\Psi_{n-1}} \dots \overline{\Psi_{n-j}} i \rangle \overline{\Psi_{n-(j+1)}} \dots \overline{\Psi_0}$
pop_k	$\overline{\Psi_{n-k,q}} \overline{\Psi_{n-k-1}} \dots \overline{\Psi_0}$

It is easy to verify that every application term in the above table obeys the syntactic restriction of the safety constraint from Def. 2.11. Hence since the recursion scheme is homogeneously-typed, by Proposition 2.2, it is also incrementally-bound. \square

References

- [1] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. Safety is not a restriction at level 2 for string languages. Technical report, University of Oxford, 2004.
- [2] W. Blum. *The Safe Lambda Calculus*. PhD thesis, University of Oxford, 2009. <https://ora.ox.ac.uk/objects/uuid:537d45e0-01ac-4645-8aba-ce284ca02673>.
- [3] W. Blum. Type homogeneity is not a restriction for safe recursion schemes (note). June 2009.
- [4] W. Blum and C.-H. L. Ong. The safe lambda calculus. In S. R. D. Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2007.
- [5] C. Broadbent. A proof of Blum's conjecture, June 2009.

- [6] W. Damm. The IO- and OI-hierarchy. *TCS*, 20:95–207, 1982.
- [7] J. G. de Miranda. *Structures generated by higher-order grammars and the safety constraint*. D.Phil thesis, University of Oxford, 2006.
- [8] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible push-down automata and recursive schemes. extended version (59p), November 2006.
- [9] M. Hague, A. S. Murawski, C.-H. L. Ong, and O. Serre. Collapsible push-down automata and recursive schemes. *LICS*, pages 452–461, 2008.
- [10] T. Knapik, D. Niwiński, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FOSSACS’02*, pages 205–222. Springer, 2002. LNCS Vol. 2303.
- [11] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of IEEE Symposium on Logic in Computer Science.*, pages 81–90. Computer Society Press, 2006. Extended abstract.